Distributed Trust: Supporting Fault-tolerance and Attack-tolerance

Fred B. Schneider[1]

Department of Computer Science
Cornell University
Ithaca, New York 14853

Lidong Zhou

Microsoft Research
1065 La Avenida
Mountain View, California 94043

January 20, 2004

[1]Supported in part by ARPA/RADC grant F30602-96-1-0317, AFOSR grant F49620-03-1-0156, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Material Command USAF under agreement number F30602-99-1-0533, National Science Foundation Grant 0205452, and grants from Intel Corporation and Microsoft Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

Abstract

Fault-tolerance and attack-tolerance are crucial for implementing a trustworthy service. An emerging thread of research investigates interactions between fault-tolerance and attack-tolerance—specifically, the coupling of replication with threshold cryptography for use in environments satisfying weak assumptions. This coupling yields a new paradigm known as distributed trust, which is the subject of this paper.

1 Introduction

A *trustworthy* service must operate as intended despite failures and attacks. Implementing the necessary *fault-tolerance* and *attack-tolerance* can be a real challenge. For one thing, separation of concerns does not apply, because approaches to implementing fault-tolerance can reduce a system's attack-tolerance. An example is n-fold replication of a secret s, which adds fault-tolerance and improves the availability of s but does so by increasing from 1 to n the number of sites that must resist attacks to preserve the confidentiality of s.

Other interactions also exist between schemes for achieving fault-tolerance and those for attack-tolerance. One goal of this article is to give a principled account of the landscape. A second goal is to outline a general approach for building trustworthy services from untrustworthy components, thereby providing a snapshot of an emerging research thread, which incorporates attack-tolerance not as an afterthought to fault-tolerance but as an integral part of a distributed system's architecture and protocols.

2 Replication Caveats

The basic recipe for implementing a fault-tolerant service [19, 27] is well known:

1. Start with a server, structured as a deterministic state machine, that reads and processes client submitted *requests* which are the sole means to change the server's state and/or produce an output.

2. Run replicas of that server on distinct hosts in a distributed system.

3. Employ a replica-coordination protocol to ensure that all non-faulty server replicas process identical sequences of requests.

Correct server replicas thus produce identical outputs for each given client request. Moreover, the majority of the outputs produced for each request will come from correct replicas provided (i) at most f replicas execute on faulty hosts, and (ii) the service comprises at least $2f+1$ server replicas. So we succeed in implementing a service that tolerates at most f faulty hosts by taking as the outputs of the service the responses produced by a majority of the server replicas.

Implicit in this basic recipe are some caveats. The first is a presumption that host failures are independent. Empirical observations from real systems confirms that this presumption is often a realistic approximation. Attacks are a different

story. An adversary that disrupts one replica will probably be able to exploit the same vulnerability at other replicas and disrupt them as well. So although having $2f + 1$ replicas might tolerate up to f faulty hosts, all $2f + 1$ replicas might succumb to a single attack. Adding attack-tolerance to fault-tolerance requires rethinking assumptions about independence, with (as we shall see) a profound effect on the system's design.

A second caveat is implicit in step 3 of the basic recipe. Replica-coordination protocols can exist only for certain computational models or, equivalently, in systems satisfying certain assumptions [11]. Assumptions are a double-edged sword. On the one hand, the system builder's job is simplified when stronger assumptions are made about environments in which a system will run. On the other hand, the adversary's job is also simplified—stronger assumptions are easier to invalidate through attacks. And once an assumption on which the system depends has been invalidated, correct system operation is no longer guaranteed so the adversary has succeeded.

Particularly problematic are the assumptions about process execution speeds and message-delivery delays that define the *synchronous model* of distributed computation. Most replica-coordination protocols assume this model, but denial-of-service attacks invalidate timing assumptions and, therefore, replica-coordination protocols based on the synchronous model are vulnerable to denial-of-service attacks. So weaker computational models must be adopted in supporting attack-tolerance, and those weaker computational models, in turn, affect what replica coordination is possible.

A final caveat accompanying our basic fault-tolerance recipe is related to confidentiality. Even services that do not operate on confidential data are likely to store cryptographic keys so that messages can be authenticated. Confidentiality of these keys must be maintained. As discussed in §1, an adversary bent on learning confidential information benefits when a copy of that information is stored at many server replicas. Straightforward replication, the mainstay of fault-tolerance, is thus antithetical to attack-tolerance if confidentiality is sought.

So the road from fault-tolerance to trustworthy services might start with replication but doesn't end there. The three caveats just identified suggest changes needed to the basic recipe in order to achieve attack tolerance. The remainder of this article sketches a design space for implementing those changes, with the caveat about independence of hosts discussed in §4, replication-management for weaker computational models discussed in §5, and confidentiality of service data discussed in §6.

3 Compromise and Recovery

Two general types of components are involved in building trustworthy services: processors and channels. Processors serve as hosts; channels enable these hosts to communicate.

A *correct* component only exhibits intended behavior; a *compromised* component can exhibit other behaviors. Component compromise is caused by failures and/or attacks. We make no assumption about the behavior of compromised components (the so-called Byzantine failure model), but we do conservatively assume that a component C compromised by a successful attack is then controlled by the adversary, with any secrets stored by C becoming known to the adversary.

Channel Compromise. Secrets an adversary learns by compromising one component might facilitate the subsequent compromise of other components. For example, a correct channel protects the confidentiality, integrity, and authenticity of messages it carries. This functionality is typically implemented cryptographically, with keys stored not by the channel itself but at the hosts serving as the channel's endpoints. An attack that compromises a host thus yields secrets that then allow the adversary to compromise all channels attached to the host.

Because channels are typically implemented by the hosts serving as the channel's endpoints, trustworthiness for a service is often specified solely in terms of which or how many host compromises can be tolerated. The possibility of channel compromise separate from host compromise is thus ignored. This approximation, which we also adopt in this paper, is sensible when the network topology provides several physically independent paths to each host, because then the channel connecting a host is unlikely to fail independent of that host.

Host Recovery Protocols. The system builder has little control over how and when a component transitions from being correct to being compromised. A *recovery protocol* provides the means to reverse such transitions. For a faulty component, the recovery protocol might involve replacing or repairing hardware. For a component that has been attacked, the recovery protocol must not only evict the adversary—perhaps by restoring code (ideally with newly discovered vulnerabilities patched) from clean media and by reconstituting state (perhaps from other servers)—but it must also replace any secret keys the adversary might have learned.

The reason to execute a recovery protocol after detecting a failure or attack

is obvious. Less obvious are the benefits that accrue from executing a recovery protocol periodically, even though no compromise has been detected [15]. To wit, such *proactive recovery* defends against undetected attacks and failures by transforming a service that tolerates t compromised hosts over its entire lifetime into a system that tolerates up to t compromised hosts during each *window of vulnerability* delimited by executions of the recovery protocol. The details of coordinating the servers without making synchrony assumptions might be tricky, but the benefits are substantial: with proactive recovery an adversary that cannot compromise $t + 1$ hosts within a window of vulnerability is foiled and forced to begin anew on a system with all defenses restored to full strength.

Denial-of-service attacks can slow execution, thereby lengthening the window of vulnerability and increasing the interval available to perpetrate an attack. Whether such a lengthened window of vulnerability is significant depends on whether it affords the adversary an opportunity to compromise more than t servers during the window. But whatever the adversary, systems with proactive recovery are inherently more resilient than those without it, simply because proactive recovery affords the opportunity for servers to recover from past compromises.

Service Keys: Refresh and Scalability

A client, after making a request, awaits responses from servers. When the compromise of up to t servers must be tolerated, the same response received from fewer than t servers cannot be considered correct. But if the response is received by $t + 1$ or more servers then it was necessarily produced by a correct server. So sets of $t + 1$ servers together *speak for* the service, and clients require some means to identify when equivalent responses have come from $t + 1$ distinct server replicas.

One way to ascertain the origin of responses from correct servers is to employ digital signatures. Each server's response is digitally signed using a private key known only to that server; the receiver validates the origin of a response by checking the signature using that server's public key. A server's private key thus speaks for that server. Less expensive schemes, involving message authentication codes (MAC) and shared secrets, have also been developed [9].

The use of secrets—be it private keys or shared secret keys—for authenticating server replicas to clients impacts the scalability of a service that employs proactive recovery. This is because new secrets must be selected at the start of each window of vulnerability, and clients must then be notified of the changes. If the number of clients is large, then this communication is expensive and the resulting service

ceases to be scalable.

To build a service that is scalable, we seek a scheme whereby clients need not be informed of periodic changes to server keys. Since sets of $t+1$ or more servers speak for the service, a client could identify a correct response from the service if the service is provided with some way to digitally sign responses exactly when a set of servers that speak for the service agree on that response:

TC1: Any set of $t + 1$ or more server replicas can cooperate and digitally sign a message on behalf of the service.

TC2: No set of t or fewer server replicas can contrive to digitally sign a message on behalf of the service.

TC1 implies that information held by $t + 1$ or more servers enables them to together construct a digital signature for a message (namely, for the service's response to a request), whereas TC2 implies that no coalition of t or fewer servers has enough information to construct such a digital signature. In effect, TC1 and TC2 characterize a new form of private key for digital signatures associated with the service (rather than with the individual servers). This private key speaks for the service but is never entirely materialized at individual servers comprising the service.

A private key satisfying TC1 and TC2 can be implemented using secret sharing. An $(n, t + 1)$ *secret sharing* for a secret s is a set of n random *shares* such that (i) s can be recovered with knowledge of $t + 1$ shares, and (ii) no information about s can be derived from t or fewer shares. Not only do protocols exist to construct $(n, t + 1)$ secret sharings [28, 2] but *threshold digital signature* protocols [3, 10] exist that allow a digital signature to be constructed for a message from $t + 1$ *partial signatures*, where each partial signature is computed using as inputs the message and only a single share of the private key. Thus, TC1 and TC2 can be implemented by using $(n, t + 1)$ secret sharing and dividing the service private key among the server replicas—one share per replica—and then having servers use threshold digital signatures to collaborate in signing responses.

Just like with ordinary secret keys or private keys, attack-tolerance improves if proactive recovery periodically refreshes shares of the service's signing key. In particular, this defends against so-called *mobile adversaries* [24] which attack, compromise, and control one server for a limited period before moving to the next. Over time, a mobile adversary might compromise $t + 1$ servers, obtain $t + 1$ shares, and thus be able to speak for the service, generating correctly signed bogus service responses.

The defense against mobile adversaries is, as part of proactive recovery, for servers periodically to (i) create a new and independent secret sharing for the service private key, and then (ii) delete the old shares, replacing them with the new shares. Because the new and old secret sharings are independent, a mobile adversary cannot combine new shares and old shares in order to obtain the service's signing key. And because old shares are deleted when replaced by new shares, a mobile adversary must compromise more than t servers within a single window of vulnerability in order to succeed.

Protocols to create new, independent sharings of a secret are called *proactive secret sharing* protocols and have been developed both for synchronous systems [15] and for asynchronous systems [5, 34]. The problem is a bit tricky to implement:

- The new sharing must be computed without ever materializing the shared secret at any server. (A server that materialized the shared secret, if compromised even momentarily, could be in danger of revealing the service's signing key to the adversary.)

- The protocol must work correctly in the presence of as many as t compromised servers, which might provide bogus shares to the protocol.

Server Keys: Refresh

Secure communications channels between servers is required for proactive secret sharing and the various other protocols servers execute. Since keys used to implement a secure channel are stored by the hosts at the endpoints of that channel, we conclude that, not withstanding the use of secret sharing and threshold cryptography for service private keys, there will be other cryptographic keys stored at servers. If these other keys can be compromised then they too must be refreshed during proactive recovery. Three classes of solutions have been proposed.

Tamper-proof Hardware. Although not in widespread use today, special-purpose, tamper-proof hardware that stores keys and performs cryptographic operations (e.g., encryption and decryption) does exist. Moreover, interest seems to be increasing, with industry groups like TCG (Trusted Computing Group), its predecessor TCPA (Trusted Computing Platform Alliance), and recent product announcements from Intel and Microsoft joining IBM's long-standing efforts.

The special-purpose, tamper-proof hardware is designed so that, if correctly installed, it will not divulge keys even if the software on the attached host has been compromised. When keys stored by a server cannot be revealed, there is no reason to refresh them. So, storing server keys in tamper-proof hardware attached to a server eliminates the need to refresh server keys as part of proactive recovery.

Tamper-proof hardware does not prevent a compromised server from performing cryptographic operations for the adversary. The adversary might, for example, cause the server to generate signed or encrypted messages for later use (in attacks). To detect such bogus messages, the tamper-proof hardware could maintain an integer counter in stable memory (so the counter's value will persist across failures and restarts). The counter is incremented every time a new window of vulnerability starts; and the current counter value is included in every message that is encrypted or signed using the tamper-proof hardware. A server can now discard any message it receives that has a counter value too low for the current window of vulnerability.

Off-line Keys. Here, new keys are distributed using a separate secure communications channel the adversary cannot compromise. This channel will typically be implemented cryptographically by using secrets that are stored and used in an off-line stand-alone computer, thereby ensuring inaccessibility to a network-borne adversary. For example, an administrative public/private key pair could be associated with each server H. The administrative public key $\hat{K}_H$ is stored in ROM on all servers; the associated private key $\hat{k}_H$ is stored off-line and is known only to the administrator of H. Each new server private key k_A for a host A would be generated off-line. The corresponding public key K_A would then be distributed to all servers by including K_A in a certificate signed using the administrative private key $\hat{k}_A$ of server A.

Attacks Awareness. Instead of relying on a full-fledged tamper-proof co-processor, a scheme in Canetti and Herzberg [8] uses non-modifiable storage (e.g., ROM) to store a special service-wide public key, whose corresponding private key is shared among servers using an $(n, t+1)$ secret sharing. To refresh its server key pair, a server H generates its new private/public key pair, signs the new public key using the old private key, and then requests that the service endorse the new public key. Such an endorsement is represented by a certificate that associates the new public key with server H and that is signed using the special service private key.

The service private key is refreshed periodically using proactive secret sharing, thereby guaranteeing that an attacker cannot learn the service private key provided the attacker cannot compromise more than t servers in a window of vulnerability. Therefore, an attacker cannot fabricate a valid endorsement because bogus certificates are detected by servers using the service public key stored in their ROM. A server becomes aware of an attack if it does not receive a valid certificate for its new public key within a reasonable amount of time. In this case, actions are initiated to re-introduce the server into the system and remove the possible imposter of that server.

A server's receipt of two conflicting requests (signed by the victim's private key) for endorsement triggers the broadcast of an alarm. Such conflicting requests could occur if an attacker compromises a server, obtains the server's private key in the previous window of vulnerability, and sends a key refresh request signed by that private key. To impede denial-of-service attacks based on false alarms, an alarm message is ignored unless it is accompanied by the conflicting (signed) endorsement request messages as evidence. Manual interventions are required when any server detects an alarm.

4 Eluding the Processor Independence Caveat

We elude the processor independence caveat to the extent that actions—attacks or host failures—cannot cause multiple hosts to become compromised. For example, independence is reduced when

- hosts employ common software (and thus all replicas could be compromised by the same attacks),
- hosts are operated by the same organization (because a single maleficent operator could then access and compromise more than a singled host), or
- hosts rely on a common infrastructure, such as the name servers or network of routers used to support their communications, since the compromise of that infrastructure violates an assumption needed for the hosts to function.

One general way to characterize the trustworthiness of a service is by describing which sets of components could together be compromised without disrupting correct operation of the service. Each vulnerability V partitions server replicas into groups, where replicas in a given group share that vulnerability. For instance,

there exist attacks that compromise server replicas running Linux but not those running Windows (and *vice versa*), which leads to a partitioning according to operating system; and the effects of a maleficent operator are likely localized to server replicas under that operator's control, which leads to a partitioning according to system operator.

Sets of a system's servers whose compromise must be tolerated can be specified using an *adversary structure* [16]. This is a set $\mathcal{A} = \{S_1, \ldots, S_r\}$ whose elements are sets of system servers an adversary is assumed able to compromise during the same window of vulnerability. A trustworthy service is then expected to continue operating as long as the set of compromised servers is an element of $\mathcal{A}$. Thus, the adversary structure $\mathcal{A}$ for a system intended to tolerate attacks on the operating system would contain sets S_i whose elements are servers all running the same operating system.

When there are n server replicas and $\mathcal{A}$ contains all sets of servers of size at most t, the result is known as an (n, t) *threshold* adversary structure [28]. The basic recipe in §2 for implementing a fault-tolerant service involves a threshold adversary structure, as does much of the discussion throughout this article. Threshold adversary structures correspond to systems in which server replicas are assumed to be independent and equally vulnerable. They are, at best, approximations of reality. The price of embracing such approximations is loss of fault- or attack-tolerance, because now single events might actually compromise all of the servers in some set that is not an element of the adversary structure.

Although threshold adversary structures are not accurate models of reality, protocols designed for threshold adversary structures frequently have straightforward generalizations to more-general adversary structures that are. What is less well understood is how to identify an appropriate adversary structure, since doing so requires determining what common vulnerabilities exist. Today's systems generally employ commercial off-the-shelf (COTS) components wherever possible, which for commercial reasons restrict access to internal details useful for identifying common vulnerabilities. Those hoping to formulate adversary structures for a system are thus handicapped.

Independence by Avoiding Common Vulnerabilities

Eliminating software bugs eliminates vulnerabilities and thus eliminates common vulnerabilities that would thwart independence of replicas. The construction of bug-free software is quite difficult, however. So instead we exploit another means to increase replica the independence: diversity. In particular, the basic recipe in

§2 for fault-tolerance through replication does not require that server replicas be identical in either their design or their implementation—only that all server replicas implement the same deterministic state machine. That is, different replicas must produce equivalent responses for each given request.

Such diversity can be obtained in three ways:

- Develop multiple server implementations. This, unfortunately, can be expensive. The cost of all facets of system development are multiplied because each replica now has its own design, implementation, and testing costs. In addition, interoperation of diverse components is typically more difficult to get working, not withstanding the adoption of standards. Finally, experiments have shown that distinct development groups working from a common specification will produce software having the same bugs [18].

- Employ pre-existing diverse components that have similar functionality and write software wrappers so that all implement the same interface and the same state machine behavior [27, 26]. One difficulty here is procuring diverse components that do have the requisite similar functionality. Some operating systems (e.g., BSD UNIX vs. Linux) have multiple, diverse implementations but other operating systems do not; and application components we use in building a service are unlikely to have multiple diverse realizations. A second difficulty arises when components do not provide access to internal non-deterministic choices they make during execution (e.g., for creating a handle that will be returned to a client), since now writing the wrapper can be quite difficult [26].

- Introduce diversity automatically during compilation, loading, or in the runtime environment [12, 32]. Code can typically be generated and storage allocated in any number of ways for a given high-level language program; making choices in producing different executables introduces a measure of diversity. Different executables for the same high-level language program are still implementations of the same algorithms, though, so executables obtained in this manner will continue to share any flaws in those algorithms.

The third of these approaches—automatic introduction of diversity—holds great promise because it allows repeatedly changing each replica during system operation, perhaps as part of proactive recovery. Thus, not only would different replicas have different vulnerabilities, but the vulnerabilities at any given host would change periodically. So an attack that succeeds during one window of vulnerability against a given host might not work during the next. Termed *proactive*

obfuscation, the infrastructure to support this approach is currently being built by Schneider and collaborators at Cornell.

Finally, note that because state machine replicas must implement the same interface, these replicas cannot be completely independent. A service that supports an operation whose semantics has a vulnerability will exhibit that vulnerability with or without state machine replication. Defining a service interface and operations whose semantics cannot be abused is thus of paramount import for building a trustworthy service.

5 Embracing the Replica Coordination Caveat

In the basic recipe of §2, not only must state machine replicas exhibit independence but all correct replicas must reach consensus about the contents and ordering of client requests. Therefore, the replica-coordination protocol must include some sort of *consensus protocol* [25] to ensure that

- all correct state machine replicas agree on each client's request, and
- if the client sends the same request R to all replicas then R is the consensus they reach for that request.

This specification involves both a safety property and a liveness property. The safety property prohibits different replicas from agreeing on different values or orderings for any given request; the liveness property stipulates that an agreement is always reached.

Consensus protocols exist only for systems satisfying certain assumptions [11], and in particular deterministic consensus protocols do not exist for systems having unboundedly slow message delivery or process execution speeds. This limitation arises because to reach consensus in such an *asynchronous system*, participating state machine replicas must distinguish between (i) those replicas that have halted (due to failures) and thus should be ignored and (ii) those replicas that, though correct, are executing very slowly and thus cannot be ignored.

The impossibility of implementing a deterministic consensus protocol in asynchronous systems leaves three options.

Option I: Abandon State Machine Replication. Instead of replicating a state machine that directly implements the service semantics, we might instead implement and replicate a server that only supports read and write operations on

storage servers. These servers are then organized as a *quorum system*, and the desired trustworthy service is implemented on top.

To constitute a quorum system, the storage servers are associated with groups; each read or write operation is executed on all servers in some group. Moreover, these groups are defined so that pairs of groups intersect in one or more servers—the effect of a write operation is thus seen by any subsequent read. Various quorum schemes differ in the size of the intersection of two quorums. For example, if faulty processors simply halt then as many as f faulty processors can be tolerated by having $2f + 1$ processors in each group and $f + 1$ in the intersection. If faulty processors can exhibit arbitrary behavior then a *Byzantine quorum system* [22], involving larger groups and a larger intersection, is required.

A consensus protocol is not needed for performing operations in a quorum system. So the impossibility result of Fischer *et al.* [11] does not apply, and protocols for read and write operations with quorums can be implemented in asynchronous systems. Unfortunately, some services have semantics that cannot be implemented using quorums of storage servers. Problematic are services supporting operations that atomically update the service's state, such as operations that involve reading the state, doing some computation, and then writing. The read and write are separate quorum operations, so a protocol is now needed to prevent concurrent requests from being interleaved differently (thereby causing states to diverge) at distinct servers of the quorum. That protocol to control request orderings at the distinct storage servers turns out to implement a form of consensus and thus cannot be built in asynchronous systems.

The practicality of structuring services in terms of quorums of replicated storage servers is not well understood. Various robust storage systems [21, 23, 31] have been structured in this way, but the feasibility of building a storage service from storage servers should not be surprising. Recently, a fault- and attack-tolerant certification authority, COCA [35], which does involve service operations having a read followed by a write, was implemented with quorums, suggesting quorums might have broader practical application than had been thought. Understanding the practical applicability of quorum structures for implementing richer service semantics in asynchronous systems has thus become an active area of research.

Option II: Employ Randomization. The impossibility result of Fischer *et al.* [11] does not rule out protocols that use randomization, and practical randomized asynchronous Byzantine agreement protocol have been developed. One example is the

consensus protocol of Cachin *et al.* [6], which builds on some new cryptographic primitives including a non-interactive threshold signature scheme and a threshold coin-tossing scheme; the protocol is part of the Sintra toolkit [7] developed at the IBM Zurich Research Center. Sintra supports a variety of broadcast primitives needed for coordination in replicated systems.

Option III: Sacrifice Liveness (Temporarily). A service cannot be very responsive when processes and message delivery have become glacially slow, so the liveness property of a consensus protocol might temporarily be relaxed in those circumstances. After all, there are no real-time guarantees in an asynchronous system anyway. The crux of this option, then, is to employ a consensus protocol (i) that satisfies its liveness property only while the system satisfies assumptions somewhat stronger than found in an asynchronous system but (ii) that always satisfies its safety property (so different state machine replicas still agree on requests they process).

Lamport's Paxos protocol [20] is a well known example of trading liveness for operation under the weaker assumptions of an asynchronous system. The same approach is taken in BFT, a Byzantine fault-tolerant protocol for implementing replicated state machines in an asynchronous system [9]. BFT can be seen as extending Paxos to tolerate Byzantine failures and has been used to implement BFS, a Byzantine fault-tolerant NFS service [9]. Considerable effort was devoted to engineering the BFT protocols so that they are cheap in the absence of failures or attacks, which was thought to be the common case.

6 Caveat about Server Confidential Data

Some services involve data that must be kept confidential. Unlike secrets used in connection with cryptography (*viz.* keys), such server data cannot be changed periodically as part of proactive recovery, because values now have significance beyond just being secret and could be part of computations that support the services semantics.

Information stored unencrypted on a server become known to the adversary if that server is compromised. Thus, confidential service data must always be stored in some sort of encrypted form—either replicated or partitioned among the servers. Unfortunately, few algorithms have been found that perform interesting computations on encrypted data (although some limited search operations can now be supported [29]). Even temporarily decrypting the data on a server

replica or storing it on a backup in unencrypted form risks disclosing secrets to the adversary.

One promising approach is to employ *secure multi-party computations* [14]. Much is known about what can and cannot be done as a secure multi-party computation; less is known about what can and cannot be done efficiently, and the prognosis is not good for efficiently supporting arbitrary computations (beyond cryptographic operations like decryption and signing).

Note, it is not difficult to implement a service that simply stores confidential data for subsequent retrieval by clients. The key elements of that solution have already been described. Confidential data (or a secret key to encrypt the data) is encrypted using a service public key. The corresponding private key is shared among replicas using an $(n, t+1)$ secret sharing scheme and refreshed periodically using proactive secret sharing. A copy of the encrypted data is stored on every replica to preserve its integrity and availability in face of server compromises and failures.

Two schemes have been proposed for a client to retrieve the encrypted data. Each ensures that the only data ever present at a server replica appears in encrypted form.

Re-encryption. A re-encryption protocol produces a ciphertext encrypted under one key from a ciphertext encrypted under another but without the plaintext becoming available during intermediate steps. Such protocols exist for public key cryptosystems where the private key is shared among a set of servers [17, 33]. So, to retrieve a piece of encrypted data, the service executes a re-encryption protocol on data encrypted under the service public key to obtain data encrypted under the client public key.

Blinding. A client chooses a random blinding factor, encrypts it using the service public key, and sends that to the service. The service multiplies the encrypted data by this blinding factor and then employs threshold decryption to compute un-encrypted but blinded data, which is sent back to the client. The client, knowing the blinding factor, can then recover the data from that blinded data. The e-vault [13] project employs this approach.

7 Status and Future Directions

Various systems have been built using the elements we have just outlined. These efforts are summarized in Figure 1 and Figure 2. There is clearly much to be

BFS [9]: An NFS file system implementation built using BFT. See Figure 2 for a description of the BFT toolkit.

COCA [35]: A trustworthy distributed certification authority. COCA avoids consensus protocols by using a Byzantine quorum systems. The system employs threshold cryptography to produce certificates signed by the service, using proactive recovery in conjunction with off-line administrator keys for maintaining authenticated communication links. COCA assumes the asynchronous system model.

E-Vault [13]: A secure distributed storage system. E-vault employs threshold cryptography to maintain private keys, uses blinding for retrieving confidential data, and implements proactive secret sharing. E-vault assumes the synchronous system model.

Figure 1: Systems that Employ Elements of Distributed Trust.

learned about how to engineer systems based on these elements, and only a small part of the landscape has been explored.

Fault-tolerance and attack tolerance are ultimately tied to a set of assumptions about the environment in which a system must function. Weaker assumptions should be preferred, since then there is less risk that they will be violated by natural events or an adversary's attacks. But that renders irrelevant much prior work in fault-tolerance and distributed algorithms.

First, prior work has assumed the synchronous model of computation. There are now good reason to investigate algorithms and system architectures for asynchronous models of computation: concern about denial-of-service attacks and interest in distributed computations that span wide-area networks. Second, prior work on replication has (mostly) ignored confidentiality. Yet confidentiality is not orthogonal and poses a new set of challenges, so it cannot be ignored. Moreover, because confidentiality is not a property of an individual component's states or state transitions, the usual approaches to specification and system refinement, which are concerned with what actions components perform, are irrelevant.

The system design approach outlined in this paper has been referred to as implementing *distributed trust* [4], because it allows a higher level of trust to be placed in an ensemble than could be placed in a component. There is no magic here. Distributed trust requires that component compromise be independent. To date, only a few sources of diversity have been investigated and only a subset of those have enjoyed practical deployment. Real diversity is messy and often brought about by random and unpredictable natural processes, in contrast to how

BFT [9]: A toolkit for implementing replicated state machines in an asynchronous system. Services tolerate Byzantine failures and use a proactive recovery mechanism for periodically re-establishing secure links among replicas and restoring the code and the state of each replica. BFT employs consensus protocols and sacrifices liveness to circumvent the impossibility result for consensus in an asynchronous system. For proactive recovery, BFT assumes a secure cryptographic co-processor and a watchdog timer. BFT does not provide support for storing confidential information or for maintaining a service private key that is required for scalability.

ITTC (Intrusion Tolerance via Threshold Cryptography) [30]: A toolkit that includes a threshold RSA implementation with distributed key generation and share refreshing. Share refreshing is done when instructed by an administrator. No clear system model is provided, but the protocols seem to be suitable for use in an asynchronous system.

Phalanx [23]: Middleware for implementing scalable persistent survivable distributed object repositories. A Byzantine quorum system allows Byzantine failures to be tolerated, even in asynchronous systems. Ramdomized protocols are used to circumvent the impossibility result for consensus in asynchronous systems. Phalanx does not provide support for storing confidential information or for maintaining confidential service keys; it also does not implement proactive recovery.

Proactive security toolkit (IBM) [1]: A toolkit for maintaining proactively secure communication links, private keys, and data storage in synchronous systems. The design employs attack-awareness approach (with ROM) for refreshing the servers' public/private key pairs.

SINTRA (Secure INtrusion-Tolerant Replication Architecture) [7]: A toolkit that provides a set of group communication primitives for implementing a replicated state machine in asynchronous systems where servers can exhibit Byzantine failures. Randomized protocols are used to circumvent the impossibility result for consensus in an asynchronous system. SINTRA does not provide support for storing confidential information or for maintaining a service private key that is required for scalability, although the design of an asynchronous proactive secret sharing protocol is documented elsewhere.

Figure 2: Toolkits for Implementing Distributed Trust.

most computations are envisaged (as a preconceived sequence of state transitions). Think about how epidemics spread (from random, hence diverse, contacts between individuals) to wipe out a population (a form of "reliable broadcast"); think about how individuality permits a species to survive or how diverse collections of species allow an ecosystem to persist.

Finally, if cryptographic building blocks, like secret sharing and threshold cryptography, seem a bit arcane today, it is perhaps worth recalling that twenty years ago, research in consensus protocols was a niche concern that systems builders ignored as impractical. Today, systems designers understand and regularly use such protocols in order to implement systems that can tolerate various kinds of failures even though hardware is more reliable than ever. The promising technologies for attack tolerance, such as secret sharing and threshold cryptography, are today a niche concern. This cannot persist for long, given our growing dependence on networked computers which, unfortunately, makes us hostage not only to failures but also to attacks.

Acknowledgments

Helpful comments on earlier drafts of this paper were provided by Martin Abadi, Úlfar Erlingsson, Andrew Myers, Mike Schroeder, Gun Sirer, and Ted Wobber. Greg Roth is a collaborator on the proactive obfuscation work, and discussions with Robbert van Renesse were instrumental in developing our first prototype systems that embodied these ideas.

References

[1] B. Barak, A. Herzberg, D. Naor, and E. Shai. The proactive security toolkit and applications. In *Proceedings of the 6th ACM Conference on Computer and Communications Security (CCS'99)*, pages 18–27, Kent Ridge Digital Labs, Singapore, November 1999. ACM SIGSAC, ACM.

[2] G. R. Blakley. Safeguarding cryptographic keys. In R. E. Merwin, J. T. Zanca, and M. Smith, editors, *Proceedings of the 1979 National Computer Conference*, volume 48 of *AFIPS Conference Proceedings*, pages 313–317, New York, NY USA, September 1979. AFIPS Press.

[3] C. Boyd. Digital multisignatures. In H. Baker and F. Piper, editors, *Cryptography and Coding*, pages 241–246. Clarendon Press, 1989.

[4] C. Cachin. Distributing trust on the Internet. In *Proceedings of International Conference on Dependable Systems and Networks (DSN-2001)*, pages 183–192, Göte-

borg, Sweden, June 30–July 4 2001. IEEE Computer Society Technical Committee on Fault-Tolerant Computing, IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance, IEEE Computer Society.

[5] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 88–97. ACM, ACM Press, November 2002.

[6] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC 2000)*, pages 123–132. ACM, July 2000.

[7] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002)*, pages 167–176. IEEE Computer Society Technical Committee on Fault-Tolerant Computing, IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance, IEEE Computer Society, June 2002.

[8] R. Canetti and A. Herzberg. Maintaining security in the presence of transient faults. In Y. Desmedt, editor, *Advances in Cryptology—Crypto'94, the 14th Annual International Cryptology Conference, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 425–438, Berlin, Germany, 1994. Springer-Verlag.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[10] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *Advances in Cryptology—Crypto'89, the 9th Annual International Cryptology Conference, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 307–315, Berlin, Germany, 1990. Springer-Verlag.

[11] M. J. Fischer, N. A. Lynch, and M. S. Peterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, April 1985.

[12] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 67–72, Cape Cod, MA, May 1997. Computer Society Press.

[13] J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1–2):363–389, July 28 2000.

[14] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game. In *Proceedings of the 19th Annual Conference on Theory of Computing, STOC'87*, pages 218–229, New York, NY USA, May 25–27 1987. ACM.

[15] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In D. Coppersmith, editor, *Advances in Cryptology—Crypto'95, the 15th Annual International Cryptology Conference, Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 457–469, Berlin, Germany, 1995. Springer-Verlag.

[16] M. Hirt and U. Maurer. Player simulation and general adversary structures in perfect multi-party computation. *Journal of Cryptology*, 13(1):31–60, 2000.

[17] M. Jakobsson. On quorum controlled asymmetric proxy re-encryption. In H. Imai and Y. Zheng, editors, *Public Key Cryptography, Proceedings of the Second International Workshop on Practice and Theory in Public Key Cryptography (PKC'99)*, volume 1560 of *Lecture Notes in Computer Science*, pages 112–121, Berlin, Germany, 1999. Springer-Verlag.

[18] J. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, SE–12(1):96–109, January 1986.

[19] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[20] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[21] B. Liskov and R. Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 29–39, Calgary, Alberta, Canada, August 1986. ACM Press.

[22] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[23] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 17th Symposium on Reliable Distributed Systems*, pages 51–58, West Lafayette, IN USA, October 20–22 1998. IEEE Computer Society.

[24] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings of the 10th Annual Symposium on Principles of Distributed Computing (PODC'91)*, pages 51–59, Montreal, Quebec, Canada, August 19–21 1991. ACM.

[25] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[26] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 15–28, Banff, Canada, October 2001. ACM.

[27] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[28] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[29] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Research in Security and Privacy*, pages 44–55, Oakland, CA USA, May 2000. IEEE Computer Society Press.

[30] T. Wu, M. Malkin, and D. Boneh. Building intrusion tolerant applications. In *Proceedings of the 8th USENIX Security Symposium*, pages 79–91, Washington, D.C. USA, August 22–26 1999. The USENIX Association.

[31] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliççöte, and P. K. Khosla. Survivable information storage system. *IEEE Computer*, 33(8):61–68, August 2000.

[32] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. Technical Report UILU-ENG-03-2207 (CRHC-03-03), Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, May 2003.

[33] L. Zhou, M. A. Marsh, F. B. Schneider, and A. Redz. Distributed blinding for ElGamal re-encryption. Technical Report TR 2004-1920, Cornell University, January 2004. Submitted for publication.

[34] L. Zhou, F. B. Schneider, and R. van Renesse. APSS: Proactive secret sharing in asynchronous systems. Technical Report TR 2002-1877, Cornell University, October 2002. Submitted for publication.

[35] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.